PAPER Special Section on New Technologies and their Applications of the Internet

FreeNA: A Multi-Platform Framework for Inserting Upper-Layer Network Services

Ryota KAWASHIMA[†a)], ***Student Member***, **Yusheng JI**[††], ***Member***, ***and*** **Katsumi MARUYAMA**[††], ***Fellow***

SUMMARY Networking technologies have recently been evolving and network applications are now expected to support flexible composition of upper-layer network services, such as security, QoS, or personal firewall. We propose a multi-platform framework called FreeNA* that extends existing applications by incorporating the services based on user definitions. This extension does not require users to modify their systems at all. Therefore, FreeNA is valuable for experimental system usage. We implemented FreeNA on both Linux and Microsoft Windows operating systems, and evaluated their functionality and performance. In this paper, we describe the design and implementation of FreeNA including details on how to insert network services into existing applications and how to create services in a multi-platform environment. We also give an example implementation of a service with SSL, a functionality comparison with relevant systems, and our performance evaluation results. The results show that FreeNA offers finer configurability, composability, and usability than other similar systems. We also show that the throughput degradation of transparent service insertion is 2% at most compared with a method of directly inserting such services into applications.

key words: *upper-layer services, transparent functions insertion, system-call interposition, multi-platform framework, SSL*

1. Introduction

As current network environments are rapidly evolving, network applications need to be able to support upper-layer (session/presentation/application-layer) services, such as security and QoS. However, these services often require some expertise for developing and operation of them. As a result, incorporating services into existing user systems tends to be an experimental work and incurs changes of user systems at the time of introduction of the services. Moreover, addition/deletion of functions is needed continually for some services like security, and users have to follow latest services timely and test their systems accordingly. Therefore, a scheme realizing *the ease of introducing/testing services* is needed in current network systems.

We focus on the core functions of many network services which are sharable by many network applications and can be offered as independent components. That is, implementation of the services can be disjoined from the core functions of the application structurally, and modification of the application is not necessary when adding/updating/deleting the services. These implementations can be applied to variety of applications, hence software assets are leveraged effectively.

In order to facilitate deploying advanced network services, we have developed a multi-platform framework called FreeNA [1], which enables services to be transparently inserted into existing applications. FreeNA hides the platform-dependent issues like API and ABI (Application Binary Interface), and also offers a unified abstraction interface to its users. That is, the users only have to select the network service to be inserted into a target application and set some parameters. This means that users can instantly validate their new network services with real applications as a test without any modification of them.

So far, many related systems are proposed for the similar purposes as in [2]–[6], however, these systems have some restrictions on performance, platform, developer-oriented behavior, usability, and flexible service composition. Comparing with those systems, FreeNA is designed to overcome the drawbacks.

In this paper, we describe the design and implementation of FreeNA including the details on how to insert network services into existing applications, how to create services in a multi-platform environment, example implementation of a service with SSL, a functionality comparison with relevant systems, and our performance evaluation results. The result shows that FreeNA offers finer configurability, composability, and usability than other similar systems. We also show that the throughput degradation of transparent service insertion is 2% at most compared to a method of directly inserting such services into applications.

The rest of this paper is organized as follows. Section 2 discusses the related work. Section 3 gives the purposes of FreeNA and examples of network services. Section 4 presents an architectural makeup of the system. We describe its implementation in Sects. 5 and 6, and manageability in Sect. 7. We discuss the evaluation of FreeNA in Sect. 8. Section 9 is used to conclude this paper as well as to introduce some of our future challenges.

Manuscript received February 18, 2009.
Manuscript revised May 25, 2009.

[†]The author is with the Department of Informatics, School of Multidisciplinary Sciences, The Graduate University for Advanced Studies (SOKENDAI), Tokyo, 101–8430 Japan.

[††]The authors are with Information Systems Architecture Research Division, National Institute of Informatics (NII), Tokyo, 101–8430 Japan.

a) E-mail: kawa@nii.ac.jp

DOI: 10.1587/transinf.E92.D.1923

*FRamework for Extending Existing Network Applications.

2. Related Work

There have been a number of proposed systems that were related to FreeNA and many of these systems hook centerpiece APIs to transparently change being sent/received data. Here, related systems are categorized by their service insertion types.

• Source code level insertion

J. Zhang et al. [7] proposed an approach to introduce dynamic adaptation to legacy systems using AOP (Aspect Oriented Programming). KLASY [8] realizes AOP functions for OS kernel using Kerninst [9]. MetaSockets [2] enables adaptive services insertion for Java applications by Adaptive Java [10].

Although these systems enable a granular program enhancement mechanism, users of these systems are largely limited to developers and can only use aspect-oriented languages that correspond to the implementation language of the application. Moreover, users must have the source code of the application. However, there are many cases when the source codes of the applications are hidden.

• Runtime System call Interposition

Interposition Agents [3] trap specified system calls using a dedicated system call to alter their behaviors. DITOOLS [4] offers customized linker and loader to rebind the symbol information. TESLA [5] and Trickle [11] use a library preloading technique to inject network functions like traffic shaping. Livepatch [12] can change process images running on Linux using a BFD library and `ptrace` system calls.

Although these can be used without the source code of the application, their mechanisms are mostly platform-dependent, and preloading technique requires troublesome tasks when inserting multiple services hierarchically (they are independent one another).

In contrast, FreeNA is designed for portable system even though FreeNA uses runtime interposition. This is achieved by separating platform-dependent parts and implementing the mechanism which can select the adequate interposition mechanism depending on the platform.

• Service Insertion with Kernel Support

A *x*-Kernel [13] enables network protocols and their chains. Stream [14] supports composable linear connections of a module within a kernel.

Although the kernel support is efficient, these system use special interfaces with applications like UPI or TLI/XTI. Hence, there are compatibility problems with existing systems. Moreover, they are not available on Windows.

• Others

Dyninst API [15] can change a process image by dynamically instrumenting/removing the code into/from the image. It is provided as an unified API regardless of ABI. Dyninst API is used by FreeNA internally.

A VTL framework [6] provides a multi-platform transparent network service insertion method, but it only supports applications running on virtual machines.

3. Overview of FreeNA

In this section, we refer to the purposes of FreeNA and its characteristics, and introduce some examples of network services. First, FreeNA is not a proxy server, virtual machine, API, or client libraries, but a normal program and runs with the target application on the same machine. It works as a middleware system between applications and the operating system.

3.1 Purposes

FreeNA is developed for application users with a certain level of knowledge, developers, operators, and researchers. They may introduce new features into existing systems, or customize behaviors of systems. However, these processes will require not only source code of applications, but also modification of applications.

Then, FreeNA enables users to transparently insert user codes into the application without the source code and any modification. This mechanism is useful to application users because they can modify their applications easily, and developers since networking services can be tested in actual environment independently from core logic of the application. This usage is also useful when developers can only access source codes partially and result in fewer bugs caused by modification of source codes at development time.

To achieve the mechanism of FreeNA, it is designed as providing following characteristics.

- **General-purposed Framework**
  FreeNA can be used with many types of network applications, such as custom applications, web applications, P2P applications, mobile applications, or network control programs for various purposes.
- **Programming-language Independence**
  FreeNA does not take into consideration what programming language is used for implementing the target application.
- **Multi-platform**
  FreeNA is designed to work on various platforms and currently runs on both Linux and Microsoft Windows operating systems.
- **User-oriented**
  Users can select the network services being inserted by using a configuration file for the application. FreeNA offers several commands for these operations. That is, users are not required to write a program to insert a service as long as the network service component is ready.

3.2 Network Services

We suppose following network services with FreeNA.

- **Compression**
  This type of services shrink data being sent before passing them to OS and stretch them at the receiver side.
- **SSL**
  This type of services provide secure Internet communication to non-SSL-compliant applications on PKI framework. Unlike Stunnel proxy [16], such services ensure end-to-end transparency and better performance.
- **TCP multiplexing**
  This type of services bring multiple TCP connections together into one connection. This is effective there is an already existing TCP connection, actual throughput will be increased by using the same connection instead of establishing a new one. Especially, this method is more useful to secure connections or tunneling protocol if dedicated tools can't be used.
- **Ad-hoc Traffic shaping**
  This type of services adjust transmission rate of data packets at user-space or control the behavior of TCP protocol by setting socket parameters like TCP_MAXSEG. Therefore, users can control bandwidth utilization in an ad-hoc way without special tools.
- **Stateful application-layer firewall**
  This type of services are different with common packet filtering based firewalls in that these services check packet content or aggregated content to inspect application-layer protocols. These services will be effective to prevent applications from buffer-overflow attacks, format string attacks, and SQL injection attacks like TCP Stream Filtering [17]. Moreover, they will enhance the security of the P2P node by inspecting suspicious contents when communicating time.
- **Mobility**
  This type of services manage connectivity of applications by migrating sessions like TESLA, or try to reduce transmitting of packets for power consumption like MetaSockets.

The goals of FreeNA are to transparently offer the above-mentioned services to the applications running on a variety of major platforms, and to provide unified and abstract interfaces to users. Above all, the applications or kernels must not be modified at all for this purpose.

In practice, there are dedicated systems that offer one of above services such as Stunnel for SSL and compression, DummyNet [18] and Trickle for traffic controlling, and Zorp [19] for application-layer firewall. Users should use dedicated lower-layer networking tools because FreeNA only supports upper-layer protocol services (It is also possible to combine both FreeNA and other tools since FreeNA does not modify any existing systems).

However, there are benefits of using FreeNA, for example, users can use many network services in an unified way on multiple platforms, customized service for the specific application. Therefore, major advantage of FreeNA is providing mechanism to users to customize and combine services easier without special support of platforms and applications. In other words, FreeNA is a kind of application/user-oriented networking tool compared with other networking tools.

Note that, although FreeNA can support variety of network services and applications, there can be cases that some services are not practical for some types of applications. Therefore, users have to consider whether supposing services are effective to their applications.

4. FreeNA Architecture

FreeNA can be used to transparently insert network services into the communication path between end-applications (See Fig. 1). For instance, when an application (sender) transmits a packet to the receiver, the packet is intercepted and processed by the services inserted by FreeNA, then it is passed to the underlying OS. At the receiver side, the services also intercept the packet from the OS and execute the opposite process; then they pass the result to the receiver application.

When inserting network services, FreeNA employs a notion of *flow handler* [5]. The flow hander takes one input data flow and several output data flows. Each network service corresponds to an instance of the flow handler. FreeNA combines the output flow of the upstream handler with the input flow of the downstream hander.

4.1 System Components

Figure 2 shows the overall architecture of FreeNA. As you can see, FreeNA is mainly composed of the *FreeNA client* and the *FreeNA server*.

The FreeNA client provides an user-interface. Users execute the client to access the FreeNA server for inserting desired services into the target application. The current user-interface is a command-based interface like a GNU debugger (GDB) or DBMS client (supported commands are listed in the Appendix C).

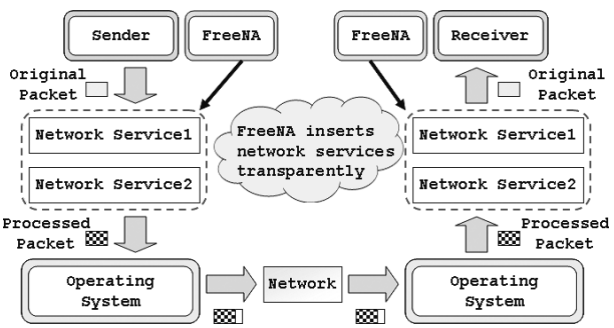


Fig. 1 Image of network service insertion.

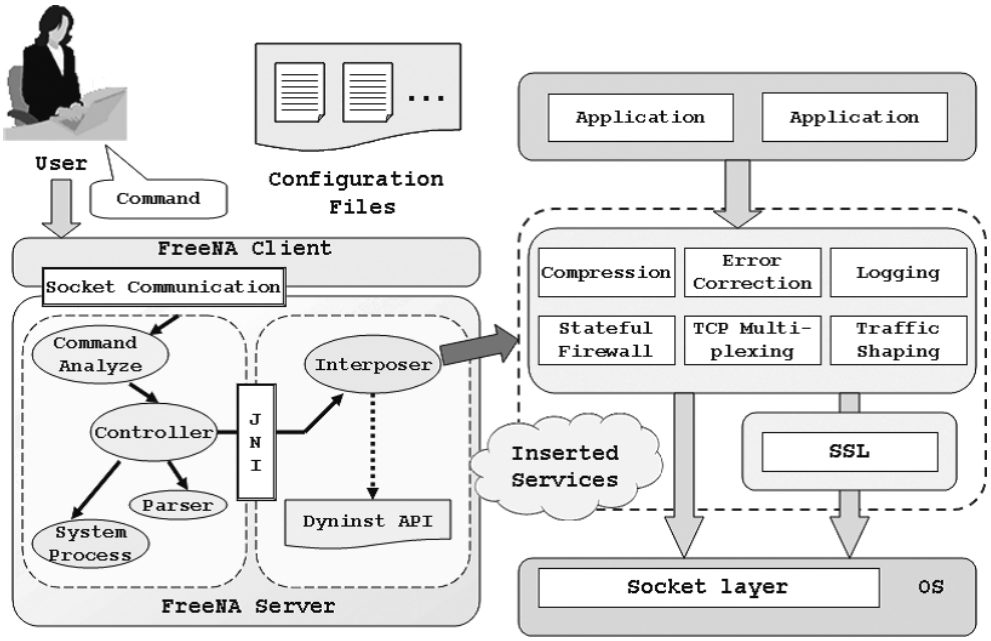


Fig. 2 Overall architecture of the FreeNA.

The FreeNA server executes invocation of the target application and inserts the specified network services based on a configuration file. In particular, an *Interposer* component invokes the application as a child process of the server and switches the destination of output data flow from socket layer to inserted network services with a Dyninst API.

4.2 The Configuration File

The configuration file must be prepared for each application to choose services and their parameters. The structure of the file is shown in Fig. 3.

A **service** tag specifies a network service, its parameters, library name, and a local using rule. Multiple services can be inserted into the same application in the order of being written. A **rule** tag (local rule) is used to specify a packet flow type by a set of conditions, such as the transport protocol, port number, and communication type (client/server). The **rule** tags also appear inside of the **using-rules** tag. These rules (global rules) are applied to all services. The rules are written in descending priority order and users can use '*' to express *all* (protocols, port numbers and so on). Note that the local rules come before the global rules.

In the case shown in Fig. 3, both *firewall* and *SSL* services are inserted when the application uses TCP, port 8080 as the server according to the global rules, and only the *SSL* service is inserted when the application uses TCP, port 8020 and 8021 according to the local rule. Services are not inserted at all in other cases.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration application="myserver">
<services>                                  • Network service
 <service name="firewall" lib="libfreena_fw.so">
  <parameter name="SQL-injection" value="ON"/>
  <parameter name="Abnormal" value="disconnect"/>
 </service>
 <service name="SSL" lib="libfreena_ssl.so">
  <parameter name="CA" value="rootcert.pem"
  <parameter name="PrivateKey" value="server.pem"/>
  <parameter name="Version" value="SSLv3:TLS"/>
   <rule use="true" service="FTP" transport="TCP"
         port="8020-8021" type="*"/>
 </service>                                 • Local rule
</services>
<using-rules>                               • Global rule
 <rule use="true" service="HTTP" transport="TCP"
       port="8080" type="server"/>
 <rule use="false" service="*" transport="*"
       port="*" type="*"/>
</using-rules>
</configuration>
```

Fig. 3 The configuration file. Users have to specify the inserting network services with parameters and insertion rules with XML-style notation.

4.3 The Network Service

FreeNA inserts network services as shared libraries into the application. The libraries provide "socket-like" functions such as `send()` and `recv()`, and they are carefully implemented so that they do not depend on other service libraries and the socket library. Therefore, FreeNA can concatenate various combinations of network services.

5. Implementation

This section describes the internal architecture of FreeNA.

First, the implementation of the FreeNA client/server is explained, then the mechanism of the service insertion and the implementation of the flow handler are explained.

5.1 FreeNA Client/Server

Since FreeNA aims to work on several platforms, most of it is implemented in Java. In Fig. 2, the whole client and the left dotted square of the server are coded in Java, and the right dotted square is implemented as a library coded in C++, because the *Interposer* is platform-dependent. The library is accessed via Java Native Interface (JNI) by the Java-coded part.

5.2 Network Service Insertion

As we mentioned in previous section, network services are implemented as independent shared libraries based on the concept of the flow handler. So, all FreeNA has to do is to bind input/output data flows of service libraries, switch from the socket function calls invoked by the application to library function calls of the uppermost service library, and have the undermost service calling actual socket functions. We subsequently explain these key mechanisms.

5.3 Flow Handler Chain

To compose the flow handler chain, we define a `service_info` C structure shown in Fig. 4. Each network service library has one corresponding `service_info` structure instance. The structure contains service parameters and insertion rule information of the service. Moreover, it contains function pointers to functions of the downstream service library and the pointers are set by the outside of the service library. Therefore, one network service can use another service even though the service library itself does not know details of another library.

```
struct service_info
{
  /* Pointer to the downstream service's one */
  struct service_info* next;

  /* Parameter information of this service */
  int    num_of_params;
  char** params;
  char** values;

  /* Service insertion rule contains effective
     port numbers and communication type */
  struct rule tcp;
  struct rule udp;

  /* Function pointers to initialization/
     finalization functions of the downstream
     service */
  void (*service_init)(struct service_info*);
  void (*service_exit)(void);

  /* Function pointers to corresponding socket-like
     functions of the downstream service */
  socket_t (*service_socket)(int, int, int);
  int (*service_bind)(socket_t, const sockaddr*,
                      socklen_t);
    ...
};
```

Fig. 4 The `service_info` structure definition. All member variables are set by a control library.

Eventually, FreeNA hierarchically inserts network service libraries between the application and the socket library. Figure 5 expresses the diagram of the hierarchy. As it can be seen in the figure, not only service libraries, but also a *Control library* and an *Interface library* are inserted together. The control library dynamically loads all underlying libraries into the application process and sets up the `service_info` structures with the configuration information passed from the FreeNA server. The `service_info` structures are formed as linked list and passed down to downstream library by `init()` functions. The interface library is inserted to access the intrinsic socket library of the platform. *SSL library* is also one of the network service libraries. However, it differs from other libraries in that SSL library is located as downmost library and used instead of interface library. We present implementation details of the library later.

By comprising FreeNA with above hierarchical architecture, arbitrary service libraries can be inserted transparently into the socket function call flow between the application and the socket library. Furthermore, FreeNA realizes more flexible function call flows.

Service insertion rules mentioned in Sect. 4.2 are used to switch call flows. When the condition is satisfied, the control library calls the service library's function. Otherwise, the control library bypasses the underlying libraries and directly calls the actual socket functions.

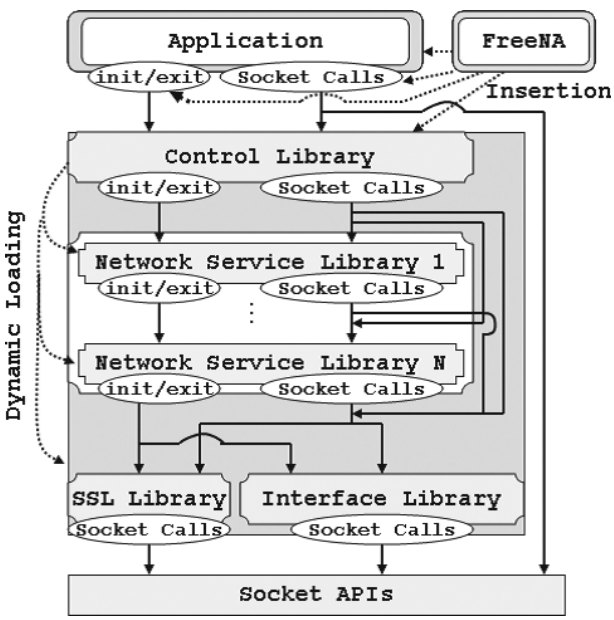


Fig. 5 Hierarchical structures of inserted services.

5.4 Socket Function Call Interposition

So far, we have explained the flow of function call through the control library, service libraries, and interface libraries. The remaining concern is that how to switch from the socket function calls invoked by the application to control library's function calls.

FreeNA leverages the runtime system-call interposition mechanism to hook socket functions. This method is suitable to realize FreeNA's mechanism than Proxy-based interposition and source code level interposition. Because, hooking of socket functions at the process image can ensure better performance and end-to-end transparency than Proxy-based interposition. Moreover, users do not need to have the source code of the application, know the internal structure of the application, nor consider the programming languages like source code level interposition.

In practice, we use the dyninst API [15] for interposition. The API provides a variety of methods for dynamically changing the runtime process image such that instrumenting/removing the CPU instructions into/from the image in an abstract manner.

Figure 6 shows a schematic process image of the application with FreeNA. First, FreeNA launches the application and loads the control library into the process image, then a `ctl_init()` and a `ctl_exit()` function of the library are embedded into the `main()` function of the application before starting. The configuration information is also embedded as the arguments of `ctl_init()`. Next, FreeNA switches the function call target from the socket library to the control library by rewriting the process image. Network service libraries and interface libraries are dynamically loaded by the control library at initialization time.

Here, since Interposer is dependent from core FreeNA server, it is possible to call another Interposer library which leverages other interposition mechanisms, such as library preloading and Detours [20]. This alternation is useful when users have platforms which Dyninst API cannot support. Instead, another mechanisms for initialization/finalization and passing configuration information are needed. Therefore, it is better to use Dyninst API for interposition whenever it supports the platform.

6. Implementation of Network Services

We now introduce the code examples of network services. We implemented compression service library and SSL service library. Compression library is a simple library that processes data at I/O operations. SSL library provides SSL/TLS compliant secure communication mechanism based on PKI framework to applications. As we can see later, service library developers can leverage existing client library, such as OpenSSL [21], libcurl [22], and Zlib [23] to implement service libraries for FreeNA.

It should be emphasized that service libraries do not provide APIs for application developers but offer socket-like interfaces. Therefore, functions of service libraries have to be implemented to conform to the purpose of socket functions.

Figure 7 shows condensed code of sending function of compression. As you can see, `service_send()` function has same interface with normal `send()` function. Data passed from the upstream library is compressed and passed down to the downstream library using `service_info` structure of this library.

Likewise, we next present code of SSL service library as a more complicated example. Since SSL is associated with socket layer, we define the SSL library as the lowermost library. Table 1 shows functions of SSL library and their purposes. `service_init()` is used to initial-

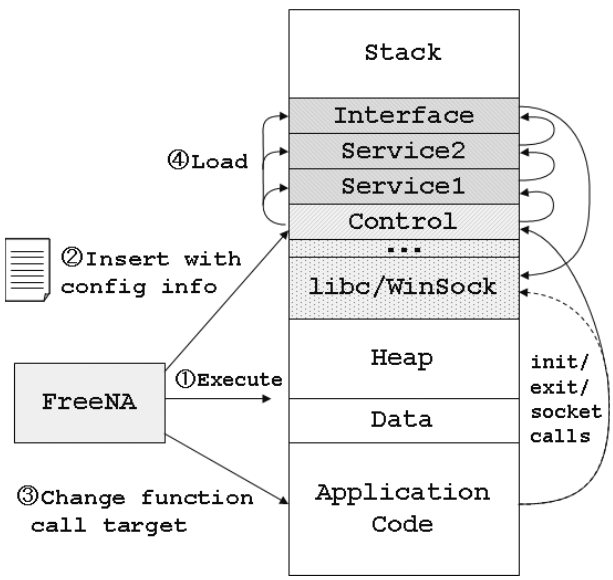


Fig. 6 Synoptic structure of process image with FreeNA.

```
ssize_t service_send( socket_t s, const char* data,
                      size_t len, int flags )
{
  ssize_t ret;
  // Service applied socket is already registered
  // according to the local rule
  if ( use_this_service( s ) ) {
    size_t new_len = BUF_SIZE - HDR_SIZE;
    // Compress 'data' and output to 'buf'
    compress( &buf[ HDR_SIZE ], &new_len, data, len );
    // Set header information (packet length)
    set_packet_length( buf, new_len );
    // Pass down 'buf' using 'service_info' structure
    // of this library
    ret = info->service_send( s, buf, new_len +
                              HDR_SIZE, flags );
  }
  else {
    // Do nothing but pass down 'data'
    ret = info->service_send( s, data, len, flags );
  }
  return ret;
}
```

Fig. 7 Example of compression service library's functions.

Table 1 SSL service library's functions and their purposes.

Library functions	Purposes
service_init	Initialize a SSL environment
service_connect	Establish a SSL connection as a client based on the connected socket
service_accept	Establish a SSL connection as a server based on the connected socket
service_send	Encrypt data and send them
service_recv	Receive data and decrypt them
service_close	Disconnect the SSL connection

```
int service_connect(socket_t s, const struct
                    sockaddr* addr, socklen_t addrlen)
{
  // Call actual 'connect' socket function
  int ret = sys_connect( s, addr, addrlen );
  // Setup a SSL object
  SSL_CTX *ctx = setup_client_ctx();
  BIO* bio = BIO_new_socket( s, BIO_NOCLOSE );
  SSL* ssl = SSL_new( ctx );
  SSL_set_bio( ssl, bio, bio );
  // Make a SSL connection
  SSL_connect( ssl );
  certification_check( ssl, addr );
  // Associate SSL object with socket
  register_socket( s, ssl );
  return ret;
}

ssize_t service_send( socket_t s, const char* data,
                      size_t len, int flag )
{
  // Get the SSL object associated with the socket
  SSL* ssl = get_SSL( s );
  // Encrypt data and send them
  ssize_t n = SSL_write( ssl, data, len );
  return n;
}
```

Fig. 8 Example of SSL service library's functions.

ize a SSL environment like loading certification file, setting random number, and determining encryption methods based on parameter information. `service_connect()` and `service_accept()` establish a SSL connection based on the already connected TCP socket. `service_send()` encrypts data being sent and pass down to the socket layer. `service_recv()` gets received data from the socket and decrypts data. `service_close()` disconnects the SSL connection.

Next, we show the condensed code example of SSL service library functions at Fig. 8. Our SSL library internally leverages OpenSSL library[†]and associates the socket with SSL session object.

7. The Manageability of FreeNA

As discussed before, FreeNA consists of many components and leverages various software techniques such as system-call interposition. Therefore, it seems that FreeNA's architecture raises the threshold of the system. In this section, we discuss the manageability of FreeNA components separately and compare with other systems.

As we mentioned at Sect. 5.1, FreeNA client is implemented as a fully independent Java program for user interface. Therefore, it can be managed as the same with normal Java programs.

FreeNA server consists of Java-coded part and the independent shared library named Interposer. Java-coded part can also be managed like normal Java program. Since Interposer library just calls Dyninst API for interposition, it can be managed like a normal shared library.

Dyninst API has been developed and managed as a part of Paradyn project at Maryland University. In FreeNA system, only Interposer library of FreeNA server uses this API, and FreeNA users and network service developers do not need to know the existence of the API. Therefore, we can integrate the latest API into FreeNA system regardless of users and service developers.

Since network service libraries are implemented as independent shared libraries with our-defined interface, and they are dynamically and fully-transparently loaded into the process image, FreeNA users and service library developers do not need to be always the same. Therefore, FreeNA users can download service libraries from third parties, and manage them independently from FreeNA client/server.

There are many other systems which leverages tricky interposition techniques. FUSE [24] hooks file operation system calls to construct user-space filesystems. Although FUSE is integrated with recent Linux kernel and many distributors support it by default, the manageability of FreeNA is easy in that users can manage FreeNA in an unified way on many platforms, and all FreeNA's components run within user-space as independent modules.

Xen [25] is another system which use binary rewriting method. In contrast, FreeNA users do not need to manage patched version of applications, modification of existing systems, nor special CPU support.

8. Evaluation

In this section, we first compare the functionality of FreeNA to similar systems, then we evaluate the performance degradation of transparent service insertion with FreeNA.

8.1 Functionality Comparison

MetaSockets [2], Interposition Agents [3], DITOOLS [4], TESLA [5], and VTL [6] are also general-purposed frameworks for inserting extended functions into existing applications like FreeNA.

Table 2 presents a summarization of the functionality of each system. *Usability* is evaluated on whether users can insert prepared service libraries without programming. *Configurability* denotes whether the system offers easily-configurable features. *Selective insertion* denotes whether the service insertion/removing can be enabled by some given conditions.

[†]We customized OpenSSL library so that the library calls our prepared function instead of original socket functions to prevent recursive socket functions calling (See Appendix A).

Table 2 Functionality comparison of each systems.

System	Users		Applications			Service Components			
	U1	U2	A1	A2	A3	S1	S2	S3	S4
FreeNA	√	√	√	Native	Linux/ Windows	Library	√	√	√
MetaSockets	N/A	Limited	N/A	Java VM	–	Java Class	√	√	Limited
Interposition Agents	N/A	N/A	√	Native	Mach 2.5 (4.3BSD)	Executable	N/A	N/A	Limited
DITOOLS	Limited	Limited	√	Native	IRIX/Linux	Library	N/A	N/A	N/A
TESLA	Limited	Limited	√	Native	Linux/FreeBSD	Executable	√	N/A	√
VTL	N/A	Limited	√	Xen/ VMWare	–	Executable	N/A	N/A	√

U1: Usability, U2: Configurability
A1: Language-Independent, A2: Runtime Environment, A3: Platform
S1: Implementation Type, S2: Independently Multiple Insertion, S3: Selective Insertion, S4: Parameter Setting

While other systems impose coding tasks to their users for network service insertion, FreeNA does not require programming to its users. To realize service insertion without programming, FreeNA offers fine configuration mechanism. Therefore, FreeNA is user-oriented system rather than developer-oriented system unlike other systems. Although DITOOLS offers the configuration file, configurable items are few and not user-oriented.

FreeNA has no restriction on the programming language used to implement the application because service functions are inserted at binary level. However, MetaSockets can only be used for Java applications. In general, systems that are based on source code level service insertion have similar limitations.

Since network services can be implemented as sharable components, several implementation types are considerable. With FreeNA and DITOOLS, network services are implemented as shared libraries. Therefore, network services can be directly loaded into application's process image. With Interposition Agents, TESLA, and VTL, services are implemented as executables.

Advanced network applications may have multiple communication flows. In this case, a mechanism that allows users to insert network services separately (*selective insertion*) is imperative. FreeNA enables the selective insertion by using the global/local rules in the configuration file. MetaSockets does not support such selective insertion. Instead, they enable runtime service insertion according to the network conditions such as packet loss rate. Other systems do not support selective insertion. Even though the mechanism can be implemented within service components themselves, it incurs more complexity in service components especially multiple services are inserted.

As we can see, FreeNA has more practical usages in that following reasons.

- FreeNA can be available on multi-platforms including Windows because of its design and the portability of Dyninst API.
- FreeNA can support many applications run on major platforms because FreeNA is available on multiple platforms, applications implemented with different languages can be supported, and FreeNA does not require virtual environment.
- Users can insert the existing service component by writing the XML-formatted configuration file without any modification of existing systems.
- Users can specify services, parameters, and insertion rules within the configuration file.
- FreeNA can support multiple service components together because FreeNA leveraging the flow handler mechanism.
- Overhead of service insertion is quite small as shown later, because service functions can be directly accessed by the application code within the process image.

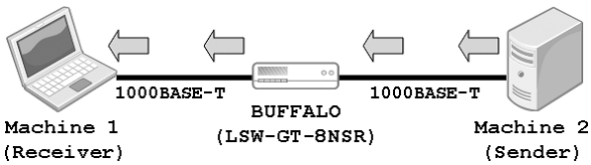


Fig. 9 Experimental network.

Table 3 Machine specifications.

Machine1	
OS	WindowsXP/Linux (2.6.18)
CPU	Intel PentiumM 1.73 GHz
Memory	512 MByte
Network	Ethernet (1000BASE-T)
Machine2	
OS	WindowsXP/Linux (2.6.18)
CPU	Intel PentiumD 2.8 GHz
Memory	4 GByte
Network	Ethernet (1000BASE-T)

8.2 Overhead Evaluation

We conducted three types of experiments to evaluate the overhead of service insertion by FreeNA. Figure 9 and Table 3 shows the experimental network and each machine's specifications. In the experiments, overhead was evaluated on both the Linux and Windows operating systems. An

application that directly calls the service functions is also evaluated for comparison purposes. The test applications were written in C++ and compiled by GNU g++/Visual Studio.NET using the best optimization option.

8.2.1 Transmission Overhead with Light-Weight Service

In the first experiment, the application on Machine1 sends 300,000 application-data with a **Null** service library, which sends data merely without any processing, and therefore the measurement directly represents the efficiency of the service insertion. Each application-data is 1024 bytes and the time is measured during all the data is transmitted to the receiver.

Table 4 shows the time for transmissions on both Linux and Windows with various numbers of null service libraries. Although the transmission times are different for different OSs, the time for FreeNA and for an application calling the service directly are almost the same (performance degradation was less than 2% at the most). Therefore, the overhead of a service insertion by FreeNA is negligible.

8.2.2 Transmission Overhead with Heavy-Weight Service

This experiment was conducted under the same conditions as the previous one except that a **Cryptography** service library and a **Compression** service library were used. The compression library uses a *Zlib* library [23] and the cryptography library uses a *Crypto++* library [26], and a *Sosemanuk* stream cipher [27] was used in the experiment. We tested the cryptography services, compression services, and both the compression and cryptography services.

The measurement results are shown in Table 5 and indicate that FreeNA does not affect the performance of the target application when using practical service libraries (performance degradation was less than 1%).

8.2.3 Overhead with Practical Usage

The third experiment was conducted using a file transfer application like FTP that uses two connections. A control connection was used to request a file and a data connection was used to transfer the file itself. The cryptography and compression service libraries were used again, and the compression service was applied to the data connection and the cryptography service was applied to both the data and the control connections. We evaluated the throughput of the client application on Machine1 while the required file was transferring through the data connection using various application-data sizes. The size of the application-data was one of the important factors affecting the system throughput, because the number of packets is inversely proportional to their size. The number of packets should generally be reduced to curtail the overheads of the packet processing.

Figure 10 shows the client's throughput on Linux and Fig. 11 shows the throughput on Windows. In Fig. 10, the throughput rapidly increased as the application-data size increased, especially for *Application-Normal*. The throughputs for these three schemes were close when the size was larger than 2048 bytes. If the compression service was used, throughput was low even though the actual application-data size was reduced by the compression. However, the throughput of the other four graphs was almost the same. Next, on Windows, the throughput of the

Table 4 Transmission time with light-weight service.

Number of calls		1	2	3	4	5
L	FreeNA	2.635	2.638	2.635	2.637	2.64
	Appli-direct	2.629	2.635	2.636	2.637	2.636
W	FreeNA	10.105	10.104	10.12	10.121	10.095
	Appli-direct	10.105	10.076	10.122	10.097	10.141

L: Linux, W: Windows

Table 5 Transmission time with heavy-weight services.

Service Name		Crypto	Comp.	Crypto+Comp.
L	FreeNA	2.641	26.242	26.785
	Appli-direct	2.641	26.197	26.721
W	FreeNA	10.404	23.574	24.141
	Applli-direct	10.389	23.625	24.128

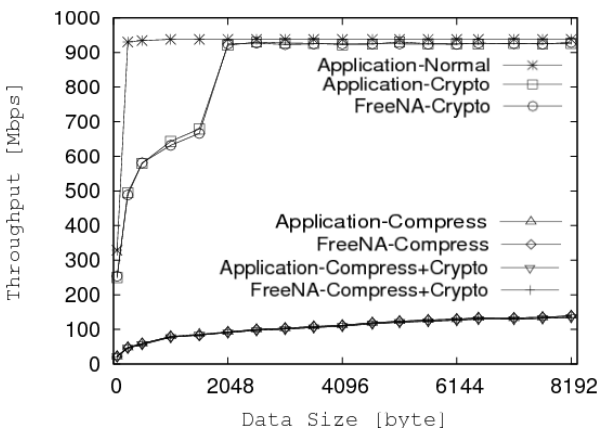


Fig. 10 Throughput of client on Linux.

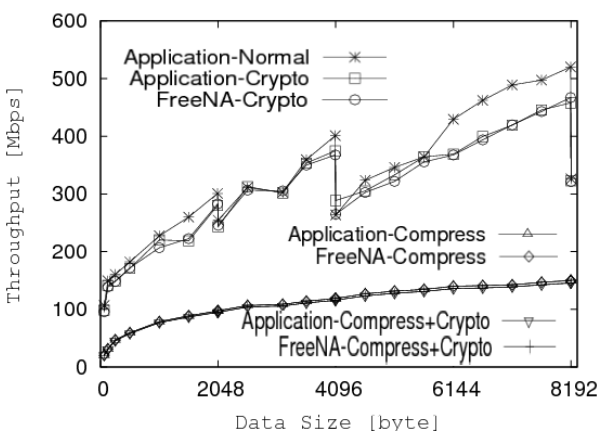


Fig. 11 Throughput of client on Windows.

normal/cryptography clients was less than that of Linux. In addition, there are some substantial performance drop points. This phenomenon may be caused by Windows or a network driver's implementation style since the test program and service libraries are the same on both Windows and Linux, and the phenomenon was not observed on Linux platform. However, there was no significant differences between FreeNA method and application-direct method. From the experiment, we can say that there seems to be no significant influence of the service insertion by FreeNA for practical usage on both platforms. Therefore, users only have to consider the overhead of processing the service functions themselves.

9. Conclusion

In this paper, we presented a multi-platform framework called FreeNA, which allows users to insert network service functions into existing applications for instant service validation with real system environment. FreeNA is composed of a client, a server, configuration files, and service libraries, and is currently available on Linux/Windows platforms.

We evaluated the functionality of FreeNA and conducted several experiments. As a result, FreeNA was able to enable more usability, portability, configurability, and practicality than other systems, and the overhead of a service insertion using FreeNA was about 1–2% at a maximum on both platforms when compared to that of an application-direct method.

Followings are advantages of FreeNA compared with other similar systems.

- FreeNA can be available on multi-platforms
- FreeNA can support many applications run on major platforms
- Users can insert the existing service component by writing the configuration file
- Users can specify services, parameters, and insertion rules
- FreeNA can support multiple service components together
- Overhead of service insertion is quite small

We are going to introduce more practical network services such as mobile service and extend FreeNA in order to dynamically insert or remove network services depending on network conditions. Each end of FreeNA may have to communicate with each other more dynamically. This approach is expected to enhance the overall performance of the system since the most suitable services and parameters can be selected based on network environments or conditions.

References

[1] R. Kawashima, Y. Ji, and K. Maruyama, "Design and implementation of multi-platform infrastructure of extensible networking functions," IEEE GLOBECOM, New Orleans, LA, USA, Nov. 2008.

[2] S.M. Sadjadi, P.K. McKinley, E.P. Kasten, and Z. Zhou, "MetaSockets: Design and operation of runtime reconfigurable communication services," Software-Practice & Experience, vol.36, no.11-12, pp.1157–1178, 2006.

[3] M.B. Jones, "Interposition agents: Transparently interposing user code at the system interface," ACM SIGOPS Operating Systems Review, vol.27, no.5, pp.80–93, 1993.

[4] A. Serra, N. Navarro, and T. Cortes, "DITOOLS: Application-level support for dynamic extension and flexible composition," Proc. USENIX Annual Technical Conference, San Diego, CA, USA, June 2000.

[5] J. Salz, A. Snoeren, and H. Balakrishnan, "TESLA: A transparent, extensible sesson-layer architecture for end-to-end network services," Proc. USITS '03, 4th USENIX Symposium on Internet Technologies and Systems, Seattle, WA, USA, March 2003.

[6] J.R. Lange and P.A. Dinda, "Transparent network services via a virtual traffic layer for virtual machines," IEEE International Symposium on High Performance Distributed Computing, Monterey, CA, USA, June 2007.

[7] J. Zhang and B.H.C. Cheng, "Towards re-engineering legacy systems for assured dynamic adaptation," International Workshop on Modeling in Software Engineering (MISE '07), Minneapolis, MN, USA, May 2007.

[8] Y. Yanagisawa, K. Kourai, S. Chiba, and R. Ishikawa, "KLASY: System for source-based binary-level dynamic weaving," J. IPSJ, vol.48, no.SIG 10(PRO 33), pp.176–188, June 2007.

[9] J. Tamches and B.P. Miller, "Fine-grained dynamic instrumentation of commodity operating system kernels," Proc. OSDI '99, pp.117–130, Berkeley, CA, USA, 1999.

[10] E. Kasten, P.K. McKinley, S. Sadjadi, and R. Stirewalt, "Separating introspection and intercession in metamorphic distributed systems," Proc. ICDCS'02, Vienna, Austria, July 2002.

[11] M.A. Eriksen, "Trickle: A userland bandwidth shaper for Unix-like systems," Proc. USENIX Annual Technical Conference, Anaheim, CA, 2005.

[12] livepatch – Live Patching for Linux, http://ukai.jp/Software/livepatch/, May 2009.

[13] N.C. Hutchinson and L.L. Peterson, "The *x*-Kernel: An architecture for implementing network protocols," IEEE Trans. Softw. Eng., vol.17, no.1, pp.64–76, Jan. 1991.

[14] D.M. Ritchie, "A stream input-output system," AT&T Bell Lab. Tech. Journal, vol.63, no.8, pp.1897–1910, Oct. 1984.

[15] B. Buck and J.K. Hollingsworth, "An API for runtime code patching," International Journal of High Performance Computing Applications, vol.14, no.4, pp.317–329, 2000.

[16] Stunnel.org, http://www.stunnel.org/, May 2009.

[17] K. Kono, T. Shinagawa, and M.R. Kabir, "Improving internet server security by filtering on TCP streams," J. IPSJ, vol.46, no.SIG 4(ACS 9), pp.33–44, 2005.

[18] IP_DUMMYNET, http://info.iet.unipi.it/~luigi/ip_dummynet/, May 2009.

[19] Zorp, http://www.balabit.com/network-security/zorp-gateway/, May 2009.

[20] Detours, http://research.microsoft.com/en-us/projects/detours/, May 2009.

[21] OpenSSL, http://www.openssl.org/, May 2009.

[22] libcurl – the multiprotocol file transfer library, http://curl.haxx.se/libcurl/, May 2009.

[23] Zlib, http://www.zlib.net/, May 2009.

[24] FUSE: Filesystem in Userspace, http://fuse.sourceforge.net/, May 2009.

[25] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," Proc. 19th ACM symposium on Operating systems principles, Bolton Landing, New York, USA, 2003.

[26] Crypto++, http://www.cryptopp.com/, May 2009.

[27] C. Berbain, O. Billet, A. Canteaut, N. Courtois, H. Gilbert, L.

Goubin, A. Gouget, L. Granboulan, C. Lauradoux, M. Minier, T. Pornin, and H. Sibert, "Sosemanuk, a fast software-oriented stream cipher," Proc. SKEW – Symmetric Key Encryption Workshop, Network of Excellence in Cryptology ECRYPT, Aarhus, Denmark, May 2005.

Appendix A: Recursive Socket Functions Calling Problem

In our approach, socket functions called by applications are replaced with functions of the control library by dyninst API. Dyninst API inserts jump instruction into process images at the beginning of socket functions' code. Therefore, when the interface library calls socket functions, control library's functions are eventually called again.

We solved this problem by using `syscall` system call on linux and customized *WinSock* library on windows. `syscall` can be used to call system calls by specifying function numbers. Customized *WinSock* library is almost the same with original *WinSock* except that function's names are slightly changed like `send` → `senX`. Since dyninst API identifies functions by their names, this approach is valid.

Appendix B: Supported Socket Functions

Common socket functions supported by FreeNA are listed in Table A· 1. FreeNA user can modify the behavior of listed functions by preparing network service libraries.

Appendix C: FreeNA's Commands

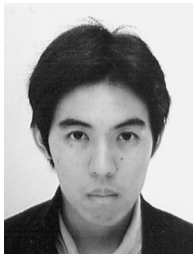
FreeNA client's commands are listed in Table A· 2.

Table A· 1 Supported socket functions (AF_INET).

`socket`	`bind`	`connect`
`listen`	`accept`	`send`
`recv`	`sendto`	`recvfrom`
`close`	`shutdown`	`getpeername`
`getsockname`	`getsockopt`	`setsockopt`

Table A· 2 List of major commands of FreeNA client.

command	description
run	Execute the specified application
dumpfile	Create the executable file
stop	Suspend the specified application
continue	Reexecute the specified application
terminate	Terminate the specified application
detach	Detach the specified application
input	Input standard-input data to the application
proclist	Show a list of running application
cd	Change the current directory
ls	List all files in the current directory
pwd	Show the path to the current directory



Ryota Kawashima was born in 1983. He received his M.S. degree from Iwate Pref. Univ. in 2007. He is a Ph.D. Candidate and has been in Graduate University for Advanced Studies (SOKENDAI) since 2007. His research areas are networking, middleware system and mobile systems. He is a member of IPSJ, JSSST, and IEEE.

Yusheng Ji received B.E., M.E. and D.E. in Electrical Engineering from The University of Tokyo in 1984, 1986 and 1989 respectively. She jointed the National Center for Science Information Systems in 1990. Currently she is an associate professor at the National Institute of Informatics, and the Graduate University for Advanced Studies. Her research interests include network architecture, resource management and performance analysis for quality of service provisioning in wired and wireless networks. She is a member of IPSJ and IEEE.

Katsumi Maruyama received B.E. and M.E. degrees in Electrical Engineering from University of Tokyo, in 1968 and 1970, respectively, and the Dr. degree in Engineering from University of Tokyo, in 1990. From 1970 to 1995, he worked at NTT. He is currently a professor at National Institute of Informatics. His research interests include distributed operating systems, concurrent object systems, real-time systems and programming languages. 1982 NTT president award. 1993 IPSJ best paper award.